Autom Softw Eng (2013) 20:463–496
DOI 10.1007/s10515-013-0125-z

A framework for semi-automated software evolution analysis composition

Giacomo Ghezzi · Harald C. Gall

Received: 30 March 2012 / Accepted: 21 March 2013 / Published online: 10 April 2013

Abstract Software evolution data stored in repositories such as version control, bug and issue tracking, or mailing lists is crucial to better understand a software system and assess its quality. A myriad of analyses exploiting such data have been proposed throughout the years. However, easy and straight forward synergies between these analyses rarely exist. To tackle this problem we have investigated the concept of *Software Analysis as a Service* and devised *SOFAS*, a distributed and collaborative software evolution analysis platform. Software analyses are offered as services that can be accessed, composed into workflows, and executed over the Internet. This paper presents our framework for composing these analyses into workflows, consisting of a custom-made modeling language and a composition infrastructure for the service offerings. The framework exploits the RESTful nature of our analysis service architecture and comes with a service composer to enable semi-automated service compositions by a user. We validate our framework by showcasing two different approaches built on top of it that support different stakeholders in gaining a deeper insight into a project history and evolution. As a result, our framework has shown its applicability to deliver diverse, complex analyses across system and tool boundaries.

1 Introduction

Until recently, historical data stored into repositories such as version control, bug and issue tracking, or mailing lists had been mostly neglected or considered a necessary byproduct of software development. However, studies have highlighted the value

G. Ghezzi (✉) · H.C. Gall
Software Evolution and Architecture Lab (s.e.a.l.), Department of Informatics, University of Zurich, Zurich, Switzerland
e-mail: ghezzi@ifi.uzh.ch

H.C. Gall
e-mail: gall@ifi.uzh.ch

of collecting and analyzing these diverse sources of data (Mockus and Votta 2000; Čubranić and Murphy 2003; Zimmermann et al. 2004). This has sparked what can be considered a "gold rush" to mine all sorts of useful information. A growing number of analysis techniques, such as static and dynamic code analyses, code clone detection, co-change analysis, bug prediction or detection of bug fixing patterns, have been devised. Yet, despite this richness, the issue of easy and straightforward integration and sharing of data produced by different analyses has been left almost entirely unaddressed.

The use and combination of different software analyses is still a challenging problem when trying to gain a deeper insight into the history of a software system. Moreover, the replication of software evolution empirical studies is negatively affected. In fact, as shown by Robles (2010), both the analyses and their results, even when available, are rarely usable for replication in an effective way. Because of this, even though software evolution research has a strong foundation on empirical studies, a systematic framework enabling replicability is still missing. We claim that this status quo severely hampers the progress of software evolution research and its soundness.

To tackle this problem, we introduced the basic concept of *Software Analysis as a Service* (Ghezzi and Gall 2008). Based on that, we devised a RESTful analysis architecture called *SOFAS* (SOFtware Analysis Services) (Ghezzi and Gall 2011). It provides the foundations for distributed analysis services, which enable a lightweight interoperability of analyses across platforms and geographical or organizational boundaries. *SOFAS* consists of three main constituents: Software Analysis Web Services (*SA-WS*), Software Analysis Ontologies (*SA-Ontos*) and a Software Analysis Broker (*SA-B*). *SA-WS* offer different software evolution analyses as standard RESTful web service interfaces. They adhere to specific meta-models and *SA-Ontos* that define and represent the data they consume and produce. The *SA-B* acts as the services manager and the interface between the services and the users. It contains a *Services Catalog* of all the registered analysis services with respect to a specific software analysis taxonomy.

In our previous papers (Ghezzi and Gall 2008, 2011) we sketched the basic idea of the approach and its architectural design. This paper presents a framework for semi-automated software analysis composition that we integrated into *SOFAS*. We explain how this composition works and describe *SCoLa*, a new language we devised to define the composition of analyses and model workflows. We introduce two concrete applications of these workflows built on top of this framework, used to investigate different aspects of the evolution of a software system. With these two applications we demonstrate the versatility of our approach in helping software evolution researchers to systematically gain a deeper and wider insight in the history and quality of software systems.

To shed light on the analysis context that we address, consider, for example, the task of getting an overview on the evolution of a specific project, e.g. Apache Tomcat,[1] using well-known indicators such as source code metrics, code clones and change coupling among the project files. To get that data, we would normally need to (1) download all the source code; (2) find and set up an appropriate metrics calculator

[1] http://tomcat.apache.org/.

(e.g. Metrics[2] or Imagix4D[3]) and a code clone detector (e.g. JCCD[4] or CCFinder[5]) and feed them the code; (3) analyze the project's SVN repository to calculate the change coupling of all its files; (4) depending on the tools used, manually interpret their results and aggregate them accordingly. This would involve dealing with different explicit and implicit meta-models and formats used to represent the data produced by the different tools. Moreover, if this process were to be repeated using any different tool, the last step would have to be redone from scratch. With our approach we can assemble a workflow that takes the project source, the version control repository URL, the clone detection strategy settings (e.g. the threshold number of tokens after which a piece of code is considered a clone) and runs the exact same process automatically. Due to the use of ontologies, the resulting data are semantically and syntactically defined, regardless of the actual metrics, code clones or change coupling analyses used (as long as they belong to the same categories). Furthermore, more analyses making use of the produced data can be added. For example, one that given the extracted source code metrics, finds all the relevant code smells, as done by Lanza and Marinescu (2005). At last, the results can be further analyzed and refined using SPARQL (or other filters and aggregators). For example, one can automatically assess if the amount of duplicated code or the value of some specific metric exceeds a certain threshold.

In Sect. 2 we give a brief overview of *SOFAS*. For a detailed description of the architectural aspects we refer to (Ghezzi and Gall 2011). Section 3 introduces *SCoLa*, our custom composition language: its main components and how its workflows are created, checked for validity, and executed. Section 4 describes how *SCoLa* is integrated and used in *SOFAS*. In Sect. 5 we briefly outline the steps needed to add a new service to *SOFAS*. In Sect. 6 we show two concrete applications of these workflows to better assess their potentiality and versatility. Section 7 gives an overview of the related work, in particular evolution analysis composition and RESTful services description and composition. We then conclude with a discussion on the strength and weaknesses of the approach and the possible future directions.

2 SOFAS

SOFAS is a RESTful architecture offering a simple yet effective way to provide software analyses. It is based on the principles of Representational State Transfer around resources on the web (as introduced by Fielding 2000). Figure 1 gives an overview of the architecture, which is made up by three main constituents: Software Analysis Web Services (*SA-WS*), a Software Analysis Broker (*SA-B*), and Software Analysis Ontologies (*SA-Ontos*). *SA-WS* expose the functionality and data of software (evolution) analyses through a standard RESTful web service interface. The *SA-B* acts as the services manager and the interface between the services and the users. It contains

[2] http://metrics.sourceforge.net/.

[3] http://www.imagix.com/products/source-code-analysis.html.

[4] http://jccd.sourceforge.net.

[5] www.ccfinder.net.

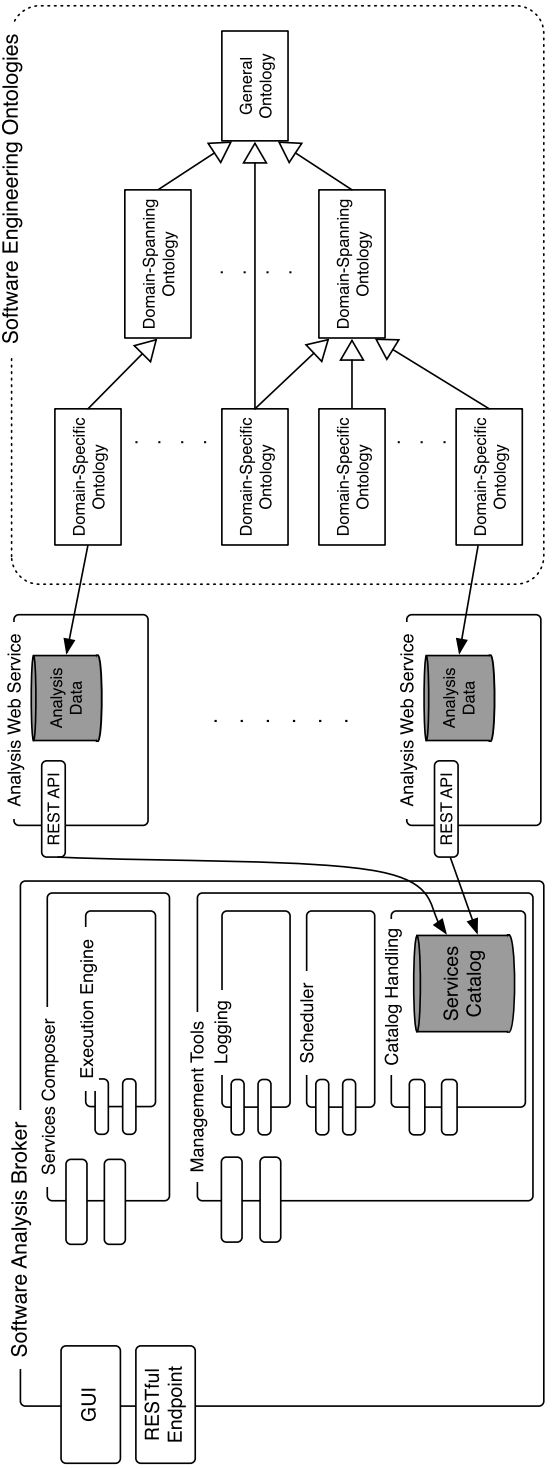

Fig. 1 *SOFAS* overall architecture

a catalog of all the registered analysis services. Ontologies define and represent the data consumed and produced by the different services. In the following, we briefly describe each of these three components.

2.1 Software analysis web services

SOFAS' purpose is to provide software analyses and the data they produce in a simple, standardized way, freeing them from specific IDEs, platforms and languages. From a user's perspective, software analyses are inherently linear and uniform in the way they work. Given some information about a software project (be it the code, its source code repository, some data already calculated by an analysis, etc.) and possible analysis calibration settings, they extract and/or calculate their specific data. Once that is completed, the results can be fetched in different, specific formats and, when needed, they can also be updated or deleted. Given these premises, RESTful services perfectly fit our needs. The main requirements and characteristics of our services are indeed some of the main inherent principles of REST.

A RESTful web service provides a uniform interface to the clients, no matter what it actually does. It is a collection of resources all identified by URIs, which can be accessed and manipulated with HTTP methods (e.g., POST, GET, PUT or DELETE). Furthermore, every message exchanged is self-descriptive as it always contains the Internet media type of the content, which is enough to describe how to process it. In our case, the analyses services boil down to simply two resources: the service itself and the individual analyses.

These analyses can be classified into three categories: (1) data gatherers; (2) basic software evolution analyses; and (3) composite software evolution analysis.

2.1.1 Data gatherers

Data gatherers work on raw data to extract evolution information from different software repositories, such as version control, issue tracking, mailing lists, or plain source code, and import it into *SOFAS* for other analyses to use it. Gathering this data can be extremely time consuming, as project histories can consist of several years of active development (e.g., Firefox version control history consists of more than 95.000 commits spread over 10 years). However, this is a vital step for any analysis, as it provides the necessary software project data to work on. At the time of writing, the following data gatherers are registered in *SOFAS*:

1. Version history importers for CVS, SVN, GIT and Mercurial. They extract the version control information comprising release, revision, and commit data from a given version control repository.
2. Issue tracking history importers for Bugzilla, Google Code, Trac, and SourceForge. They extract the issue tracking history from a given issue tracker instance.
3. GNU Mailman importer. It extracts communication data from a given GNU Mailman-based mailing list.
4. Meta-model extractors for Java and C#. They extract the static source code structure of a software project, based on the FAMIX meta-model (Tichelaar et al. 2000).

2.1.2 *Basic software evolution analyses*

Basic services exploit the data imported by one data gatherer to calculate all sorts of software evolution information: version history metrics, code metrics of specific releases/revisions, issue tracking metrics, etc. The analyses currently registered are:

1. Version history metrics calculator. It calculates several statistics from a given project version history.
2. Release meta-model extractor. It extracts the static source code structure (based on FAMIX) of one or more specific releases of a software project (written in Java or C#), given its extracted version history.
3. Code Metrics calculators. They compute some of the most common software metrics (35 as of now) of a software system. The entire list of the metrics offered, along with a brief description of them, can be found on the web.[6,7]
4. Change type distiller. Given a project version history, it extracts, for each revision, all the fine-grained source code changes of each source code file. These changes are then classified following the change types taxonomy proposed in Fluri et al. (2007).
5. Change coupling detector. It calculates the change couplings for all the files from a given version control history, as described by Gall et al. (2003).
6. Change coupling history calculator. It calculates the evolution of change couplings over the duration of a given version control history.
7. Code clones detector. It extracts the code clones from a specific version of a given version control history using JCCD.[8]
8. Code clones history calculator. It extracts the code clones from a given version control history, by regular intervals defined by the user.
9. Yesterday's Weather service. It calculates the Yesterday's Weather metric (Girba et al. 2004) from a given version control history.
10. Code ownership detector. It detects, for each file, which developers "own" it. That is the developers who should know the most about that specific file, based on how much and when they changed it. This information is extracted from a given version control history.
11. Gini coefficient calculator. It calculates the distribution of changes between the developers in a given version control history using the Gini coefficient (Gini 1912), as proposed by Giger et al. (2011).
12. Change metrics calculator. It calculates the file-level change metrics proposed by Moser et al. (2008) from a given version control history.
13. Change metrics-based defect predictor. It calculates the most defect prone files based on the change metrics calculated by the aforementioned change metrics analysis.

[6] http://habanero.ifi.uzh.ch/famixMetrics.

[7] http://habanero.ifi.uzh.ch/javaFamixMetrics.

[8] http://jccd.sourceforge.net.

2.1.3 Composite software evolution analyses

Composite services aggregate data produced by other analyses to calculate more complex and domain spanning evolution information. These are some of the analyses currently registered in *SOFAS*:

1. Issue-revision linkers. Given the issue tracking and version histories of a specific software project, they reconstruct the links between issues and the revisions that fixed them. As of now three of them exist, using the heuristics proposed by Mockus and Votta (2000), Sliwerski et al. (2005), and Fischer et al. (2003).
2. Code Disharmonies detector. It detects all the code disharmonies (Lanza and Marinescu 2005) in a software project using the code metrics extracted by the aforementioned metrics calculators.
3. Code-churn-based defect predictor. It predicts the most defect prone entities based on the combination of source code metrics calculated for specific snapshots of a given version control history. It is based on the algorithm proposed by D'Ambros et al. (2010).
4. Bug Cache defect predictor. Given the issue tracking and version histories of a specific software project and the links between them (detected by one of the aforementioned linkers), it predicts further faults, based on the algorithm developed by Kim et al. (2007).
5. Email-Source code linker. It links emails with source code given version history and mailing list information extracted by the associated data gatherers. It uses the algorithm proposed by Bacchelli et al. (2010).
6. Metrics-based defect predictor. It predicts the most defect prone entities based on the combination of source code metrics calculated by the aforementioned analyses.

2.2 Software analysis ontologies

To semantically describe the data produced by software analyses, we have developed our own family of ontologies, called *SEON* (Software Engineering ONtologies) (Würsch et al. 2012). An ontology is a formal description of the important concepts (classes of objects) identified in the domain of discourse and their relationship to one another (Gruber 1993). It provides a common vocabulary for a specific domain, which can be used to express the meta-data needed to capture the knowledge of the exchanged, shared, or reused data. Our ontologies, defined in OWL, are organized in a pyramidal structure. At the bottom of the pyramid sit ontologies describing system-specific or language-dependent concepts (e.g. Java-specific language constructs, SVN-specific versioning concepts, Jira-specific issue tracking concepts, etc.). The second layer defines domain-spanning concepts that were abstracted from system or language specifics. This layer contains concepts and relationships for version control, issue tracking, or some object-oriented programming languages like Java and C#. The top layer is comprised of higher level ontologies describing general concepts, the attributes to describe them, and the relations between the concepts. Fig-

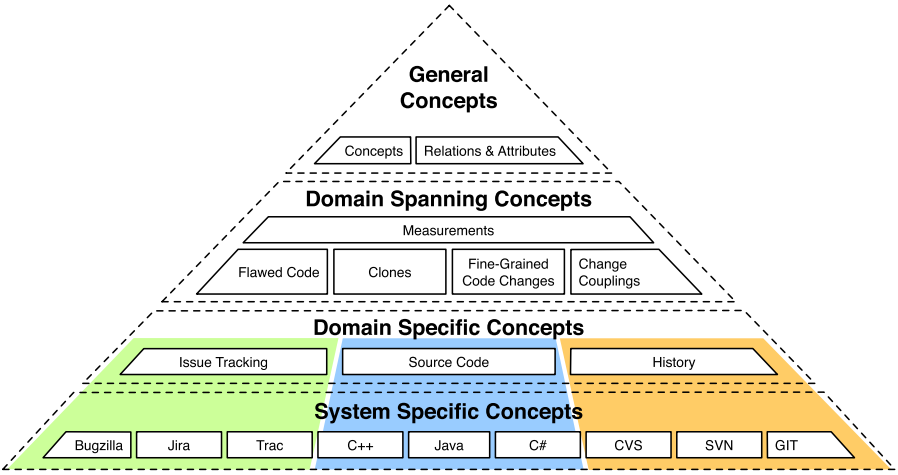

Fig. 2 A high-level view of *SEON*'s pyramid

ure 2 shows a synthetic version of SEON's pyramid. However, we refer to *SEON* 's web page[9] and Würsch et al. (2012) for a complete description of these ontologies.

3 Software analysis composition

The use of different analyses by themselves can already uncover vital information about a software project, as already shown by works such as works of Bevan et al. (2005), Nagappan and Ball (2005), Zimmermann et al. (2004). However, it is the ability to combine them into workflows, and thereby building a much broader understanding of software and its evolution, that sets the use of services apart form the current state of the art. In our case, it allows us to concatenate data gathering services with the analyses exploiting the data they produce. To effectively compose and execute these workflows, a web service composition language was required. Several of such languages have been proposed such as BPMN (2011), WSCI (2002), WS-CDL (2005) or WS-BPEL (Jordan and Evdemon 2007), with the latter emerging as the most successful and widespread.

All these languages were created with classic SOAP RPC-based services in mind. One of the main features of our solution is the use of RESTful services, which significantly differ from the former. This makes those languages hardly usable. Custom solutions, such as extending WS-BPEL to account for REST (Pautasso 2008), describing RESTful services with WSDL 2.0 (Mandel 2008) or creating new ad-hoc languages and tools (Pautasso 2009; Zhao and Doshi 2009) have been recently proposed. However, they have not really gained ground or have not been used outside theoretical case studies. This is also due to the fact that the majority of the existing

[9] www.se-on.org.

RESTful services still rely on human-oriented documentation. Besides, there has not been a concrete and widespread need to compose RESTful services yet.

Using a full-fledged approach based on WS-BPEL or on a similar solution would, in our opinion, add unnecessary complexity to something that has simplicity, uniformity, and ease of use as its main features. Furthermore, *SOFAS*' services, by being RESTful do not only have the same interface, but they also exhibit the same behavior. Analyses can be started, managed and the outcome data be fetched always in the same manner. This allows us to make several additional assumptions and simplifications in modeling how they work and how they can be composed. In particular, a workflow always consists of starting one or more analyses (an HTTP post method on the service URL), waiting for them to finish (repeatedly calling an HTTP head method on the analysis URL) and, when done, passing the URI of the results directly to waiting analyses (along with analysis specific options) or querying the results to fetch some specific data to pass to the waiting analyses—and so on, until the workflow is completed.

As a consequence, a viable solution was to develop a custom service composition language, which we called *SCoLa* (*SOFAS* Composition Language) instead of using any of the existing standards. *SCoLa* is a simplified and modified version of WS-BPEL. In the following, we will quickly go through all the fundamental components of it, assuming that the reader already has some knowledge of its parent language. Please note that a detailed, formal description of the language is beyond the scope of this paper.

3.1 An overview of SCoLa

SCoLa is intended for modeling an executable workflow of software analysis services, specifying the execution order between a number of constituent activities, the partners involved and the messages exchanged between these partners.

A workflow definition has two main sections. The *variables* section defines the data variables used in the workflow, providing their definitions in terms of XML Schema types (simple or complex). Variables allow workflows to maintain information between service calls and pass data produced by one service to another. The remainder of the description contains the actual workflow's behavior, which is, as in its parent language WS-BPEL, a kind of flow-chart. Each element in the process is called an *activity*. An activity is either *primitive* or *structured*. The primitive activity types are:

- *invoke* to invoke a service to start its specific analysis, given some input.
- *query* to query the results of an analysis with a SPARQL query passed as input and save the results into a variable.
- *exit* to terminate the entire workflow.
- *empty* to take no action.
- *save* to save content, e.g. value of variables, result of queries, etc., produced by the workflow. It is mainly used for eventual results retrieval by a user.

To enable the description of more complex structures, the following *structured activities* are provided:

- *sequence* to define an execution order.
- *flow* to define parallel execution.
- *for_each* to iterate over the results of a *query* activity or over an integer-based counter.
- *if* for conditional execution.

These structured activities can be composed and nested with each other. Furthermore, given a set of activities contained within the same *flow* or *sequence*, the execution order can further be controlled through control *links*, which allow the definition of dependencies between two activities. A target activity may only start when the source activity associated to it in the *link* has ended. Activities can be connected through links to form directed acyclic graphs.

SCoLa allows one to interact with the services only in two predefined ways: (1) starting an analysis with the *invoke* activity and (2) querying the results of an analysis with the *query* activity. This major simplification is the main difference with WS-BPEL; it is possible because of *SOFAS* services' uniform interfaces. Moreover, from *SCoLa*'s perspective all service calls are considered strictly synchronous and every service replies as soon as it is invoked. Thus, no wait or callback mechanism needs to be defined by the user. This does not mean that the actual analyses offered by the services are always instantaneous. As a matter of fact, some of them can take hours to complete (e.g., the extraction of the version history of Apache HTTPD by *SOFAS*' *GIT history service* took two hours), but the request to run them is always processed immediately. The service will then have to be regularly queried to check whether the requested analysis has completed. However, this is hidden from a *SCoLa* user. We opted to support this asynchronicity using polling rather than callback as, in our opinion, it better conforms to REST, while callbacks belong more to an RPC approach. Using polling, human users and all sort of applications can use the services in a simple and straightforward way, without having to implement any callback functionality.

At last, the language does not have any explicit exception handling. All this allowed us to greatly simplify the language without losing much expressiveness. Exceptions, asynchronicity and other low level concepts such as logging and monitoring are supported, but they are simply hidden from the user. They are always handled in the same, standard way and automatically weaved into workflows when they are translated into executable form by *SOFAS*' *Services Composer*. While extremely important for the actual success of a workflow execution, we deemed all those concepts irrelevant to the user in the case of software analysis composition. Therefore we saw no real benefit in allowing the fine tuning of them by a user and preferred simplicity and conciseness.

However, this does not mean that actual exceptions and errors are completely hidden from the users, only their handling. If workflows fail, the system will take care of tracking which service(s) failed, why it happened and communicate that to the user through its automatic exception handling. For example, the most common errors are usually caused by wrong or incomplete input to one or more services. In such cases the *SOFAS*' *Services Composer* will automatically retrieve the erroneous input from the failed service(s) state and report that to the user.

As in WS-BPEL, *SCoLa*'s workflows can either be executable or abstract. Executable workflows can be submitted 'as-is' to the *SOFAS*' *Services Composer* for execution as they contain all the necessary information. Abstract workflows, on the other hand, are only partially specified and are not intended to be executed. They hide some of the required concrete operational details, i.e. the value of workflow variables, of some of the input to be fed to the services, or even the services themselves. Calls to specific analysis services can be substituted with calls to abstract services. They are called abstract as they only exist in their WADL description and describe the features that are common to all the services belonging to a specific category of our analysis taxonomy. They are *blueprints* that all services belonging to the associated categories need to follow. In our case, the ontologies that their results and input (in case they consume data coming from other analyses) need to conform to. For example, our *GIT version history service*, can also be substituted and described by the generic, abstract *version history service* which defines the pattern and structure that any service extracting the history of a version control repository need to follow. That is, no mandatory, standard input, but output following the version control history ontology defined in our *SEON*. In terms of a *SCoLa* workflow, a call to any of these abstract 'parent' services will be the same as a call to any of its concrete children. It will simply be slightly simplified, as any service specific input is omitted and will only be added once the abstract workflow is instantiated with concrete services. However, this is enough to define a valid workflow, as all the necessary data flow and connections between the services involved is specified by the ontologies describing what they produce and consume. Even when using concrete services, attributes and workflow variables can be completely omitted or their actual value can be left undefined by using what is called an *opaque value*. They will be given the value *##opaque* which prior to execution would need to be substituted with some valid, real values given by the user instantiating the workflow.

Abstract workflows are useful to define a wide array of templates or *blueprints* at different level of abstraction. By using concrete services with *opaque values* for some of their input attributes, it is possible to define workflows that can be easily reused to analyze different projects using the same analyses. The use of abstract services, on the other hand, allows to write more generic workflows that can then be instantiated with different concrete analyses depending on the need of the moment or to have different results. For example, a workflow calculating the code clones of every release of a software project could be defined using the abstract *code clones* and *version history* services. At instantiation time the user would then need to: (1) pick the service working on the needed version control system, (2) pick the code clones detector using the desired strategy and (3) pass them the necessary settings, i.e. the URL to the repository to analyze and clone detection specific options (e.g. the tokens size). The identification and development of such abstract workflows is not addressed in this work. However, in our opinion it is a very relevant topic that deserves being investigated in detail in the future.

3.2 Workflow validation

All service input is either in the form of strings or files. The latter being files to be analyzed (e.g. source code) by the specific service; the former being both op-

tions of the analysis (e.g. URL of the version control repository to extract the history from) and the URL to data produced by other services (a.k.a. their output) to be fed to the service. All the analysis options are provided manually by the user during the composition. The syntactical correctness is checked and enforced using constraints defined in the web service description using XML schema restrictions. Restrictions are used to declare acceptable values of XML elements and attributes. For example, limit the valid content of an element to some predefined series of numbers or letters. As of now, no check is done when the input data is in the form of files. It is up to the user to provide valid, accepted files. What makes a file valid and accepted depends on the actual service being called. In fact, some services only require the syntactical validity of files, while others might also require semantical validity.

What is vital for the validity and correctness of *SCoLa* workflows is that the data flow between services is also semantically correct. That is, given any source service and the target services depending on its output, the source produces exactly the data the targets need and represents it in the right format. That means, for example, given one of our *version history services* and the *version history metrics service*, ensuring that the result of the former that feed into the latter is a version control history and is described using the proper ontology; in our case one of the *SEON* ontologies previously introduced in Sect. 2.2.

The validation and verification of service workflows has been addressed from different perspectives in several papers: from formal, model-based, verification of workflows (Foster et al. 2003; Baresi et al. 2007) to data validation (Hughes et al. 2008; Xu et al. 2010). However, all these approaches focus on classic "big web services". So far, these issues, and in particular semantical correctness and validation, have not been addressed for RESTful services yet. To mitigate this problem, we devised an ad-hoc, lightweight, validation technique based on the WADL service description. Taking inspiration from SAWSDL (Semantic Annotations for WSDL; Farrell and Lausen 2007) and SA-REST (Lathem et al. 2007), we expanded the WADL description so that any input and output of a service method may be annotated with a URI to an ontology, or an ontology class, that logically represents it, as shown in Fig. 3.

A connection between two services in a *SCoLa* workflow will be deemed semantically valid only if, based on their WADL descriptions, the output provided by the source service and the input of the target service to which it is linked (using a SCoLa control *link*) represent the same particular ontology or the same ontology concept. Obviously, only executable workflows can be fully validated. Abstract workflows can be semantically validated, as they have to declare all the links between the different services. A full semantic and syntactic validation is not possible as, being abstract, some variables and attributes are omitted or given *opaque values*.

This customization of WADL is also useful in guiding the user in the creation of workflows. In fact, given a specific service, based on its description and on the results it is declared to require and produce, it is then trivial to automatically fetch from the *Services Catalog* all the ones that produce and consume that particular data and propose them to the user. We will see this in more details in the next Section.

```
<?xml version="1.0" encoding="UTF-8">
<application xmlns="http://evolizer.org/wadl"
             xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <resources base="http://habanero.ifi.uzh.ch/releaseFamix/">
    <resource path="/analyses">
      <method id="createHistoryWithForm" name="POST">
                The method to start the analysis
        <request>
          <representation mediaType="application/x-www-form-urlencoded">
            <param name="name" style="query" type="xs:string">
                        The name of the new analysis
            </param>
1           <param name="url"
                    style="query"
                    type="xs:string"
                    ontologyClass="http://se-on.org/ontologies/domain-specific/2012/02/history.owl#Release"/>
                    The url of the release to extract the FAMIX model of
            </param>
          </representation>
        </request>
2       <response>
          <representation mediaType="text/html"
                          ontology="http://se-on.org/ontologies/domain-specific/2012/02/code.owl"/>
        </response>
      </method>
      ....
      <resource path="{name}">
        ....
      </resource>
    </resource>
  </resources>
</application>
```

Fig. 3 Snippet of the release meta-model service WADL description. The service is declared as requiring a release of a history ontology instance (*point 1*) and returning as output in the form of a static source code structure meta-model (*point 2*)

We opted for WADL instead of the more widely used WSDL to describe our services for several reasons. First of all, it is the *de facto* standard to describe RESTful web services. Even though it is not widely used yet, it is, in our opinion, the best way to describe RESTful web services. In fact, it was exactly devised to describe web applications that use the HTTP protocol to communicate in a simple yet effective manner. Even a complicated business application is described as basic operations (PUT, GET, POST, DELETE, etc.) on the resources that comprise the application's state (in our case an analysis). On the other hand, WSDL was designed to describe all sort of service interfaces using just about any protocol imaginable. This makes it a much more complex and expressive language. WSDL 2.0 also defines special HTTP bindings to describe HTTP applications in a very rich way. However, such richness comes at the cost of increased complexity. In our opinion, this added complexity would have been unnecessary for our purpose and would have outweighed the gained expressiveness. In fact, we are able to successfully describe our services with WADL without compromising accuracy and detail. In conclusion, WADL better fits our needs, as throughout our approach we favor, when possible, simplicity and ease of use over expressiveness and feature richness.

An important note to be made is that our goal was not to provide a full fledged, comprehensive, validation approach as the ones already proposed for classic web services. We developed a light-weight—yet rich enough for our needs—technique tailored to the very specific nature and structure of *SOFAS* services. It was not the main focus of our work, but rather a means to achieve our goal of a flexible framework to compose analyses.

3.3 Workflow creation and execution

SCoLa workflows are executed by the *SOFAS Services Composer*, which translates them into a concrete and executable form. They can be composed and submitted for execution in two ways: using its REST API or the *SA-B* UI. In the first case, the bare XML-based description has to be manually compiled and submitted for execution by the user to the *Services Composer* through its REST API. This option is useful for tools to automatically compose and submit their own workflows, but not for a human user. In fact it does not exploit all the benefits of *SOFAS*' guided analysis composition and the system in general. The burden of finding the right analyses, composing them in the right way, knowing the composition language, etc. is all on the user. The *SA-B* UI offers an intuitive graphical "boxes and arrows" way to compose workflows, as shown in Fig. 4. This second solution exploits *SOFAS* at its fullest, showcasing the benefits of such an integrated approach. Through its Ajax-based interface, the user can find and pick the analyses needed from the catalog browser, connect them together and provide all the necessary input all at once in the workflow editor.

Moreover, the user is guided by the system into this composition. Once a service is picked, all the ones that consume its output or that supply data needed by it are suggested. This is possible because of the custom annotations added to the services WADL descriptions we previously explained. The moment a service is selected, the composer automatically browses the catalog to fetch all the possible compatible/related services to suggest. Furthermore, thanks to this, workflows are validated as they are composed in real time and wrong combinations are exposed as soon as they are created. Using the workflow in Fig. 4 as an example, if the *change coupling detector service* is selected first, all the *version control history services* will be suggested as they produce data needed by it. By using this UI, not only the actual analyses but also their REST API are hidden from the user. She would just need some very basic information about the system to study, e.g. version control repository URL, source code, etc. and all the technicalities will be hidden behind the intuitive and simple boxes and arrows interface.

Workflows can be saved for future reuse or modification. On top of that, the *SA-B* builds and manages them as RESTful services, providing the same interface as all the other *SOFAS* services: their required input is the combination of input required by every single service with the exception of the one provided by other services in the workflow. This means that they can then be used as any another analysis service and combined, as atomic entities, with other services and workflows. *SOFAS* comes with some predefined workflows, called *analysis blueprints*, representing some of what we think are the most common and useful analyses to shed light on a software project and its evolution. These analyses are based on some of the most common software evolution analysis studies published in software engineering conferences (e.g., MSR, ICSE, FSE, etc.) and on software evolution analysis needs originating from concrete industrial case studies.

4 Software analysis broker

The *SA-B* acts as a "layer" between the services and the users, so that she does not have to interact directly with the raw services. It plays a vital role in facilitating the

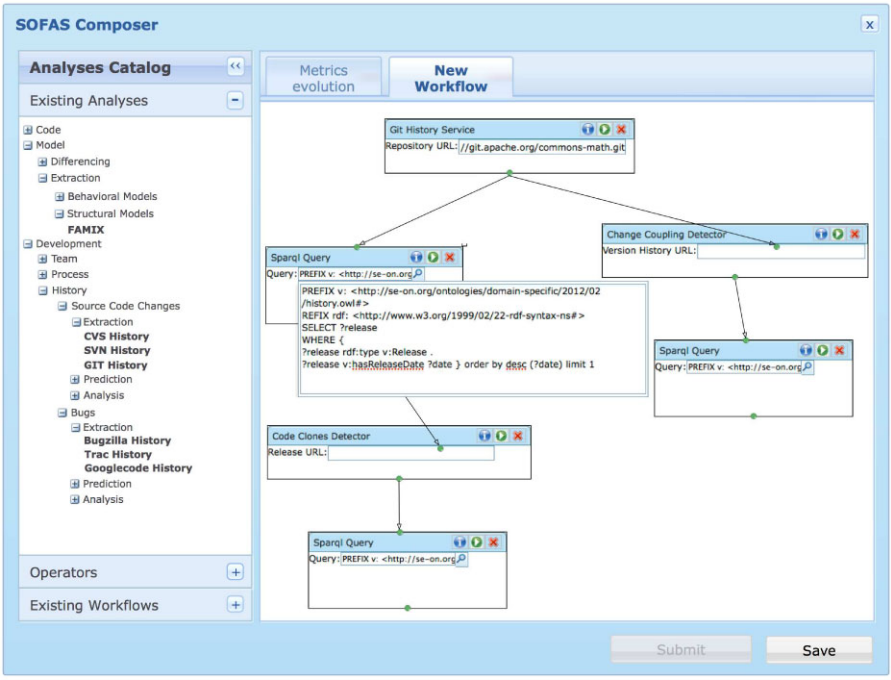

Fig. 4 Screenshot of workflow composer of the *SA-B* web UI while being used to define the Hotspot workflow described in Sect. 6.1

use of the services in an effective and meaningful way. In particular the composition and execution of the *SCoLa* workflows we just introduced in the previous section. Four main components constitute the *SA-B*: the *Services Catalog*, a series of management tools, the *Services Composer*, and a user interface.

4.1 Services catalog

The *Services Catalog* stores and classifies all the registered analysis services so that a user can discover services, invoke them, and fetch the results. We developed a software analysis taxonomy to systematically classify existing and future services. This taxonomy divides the possible analyses into three main categories: development process, underlying models, and source code. For more details we refer to the *SOFAS* website.[10]

This taxonomy is also defined as an OWL ontology. This allows us to have a very complex and rich service classification. Furthermore, SPARQL can be used to query the catalog and fetch specific services. With it, services can be queried based on what categories they belong to, on any of their attributes, on the attributes of any of the categories they belong to, etc.

[10] http://www.ifi.uzh.ch/seal/research/tools/sofas.html.

4.2 User interface

The UI is the actual access point to the *SA-B*. It consists of a web GUI, meant for human users and a series of RESTful service endpoints to be (semi)-automatically used by applications. Through it, the user can browse the *Services Catalog* to find the needed analyses, compose them and eventually run them. The user can also pick from some already predefined combinations of analysis services provided as high level analyses workflows (called *analysis blueprints*). Services can be combined into *SCoLa* workflows in a intuitive, high level and graphical "pipe and filter" fashion, as we already mentioned in Sect. 3.3.

4.3 Services composer

This component takes care of translating the workflows defined through the UI into actual, executable ones and execute them. Having the composition definition and the actual composition language decoupled, allows the user to compose services in an intuitive way, hiding the complexity and technicalities of the actual composition and orchestration. Moreover, calls to additional services, such as the ones described in Sect. 4.4 can be automatically weaved into a user-defined workflow.

4.4 Services management tools

A workflow is not just a mere collection of services called one after another. In particular, this holds when long running, asynchronous web services are involved. In order to effectively execute it, every single service needs to be logged and monitored to check if it is up and running, if it is in an erroneous state and why, if it completed a required operation, etc. We implemented a series of services that take care of implementing that as services. Calls to them can be easily and automatically weaved into a user-defined workflow by the *Services Composer*.

5 Adding a new service to *SOFAS*

In Sect. 2 we introduced *SOFAS*, the services currently registered and *SEON*, the family of ontologies used to represent the data produced and consumed by them. We then saw how these services can be combined into workflows (Sect. 3) and executed using *SOFAS*' *SA-B* (Sect. 4). In this section we quickly go through the main steps needed to create and add a new service:

1. Create the new analysis to be offered by the new service. This is the main effort and is independent from *SOFAS* and its web service-based architecture. Its difficulty can widely vary, depending on the actual analysis being offered.
2. Wrap the new analysis with a RESTful web service with an API that conforms to the one used in *SOFAS*. That means that the service has to expose two resources: the service itself and the individual analyses it produces. Each of these resources has to offer specific HTTP methods. For the complete list and explanation of these methods we refer to our previous work describing *SOFAS* in detail (Ghezzi and Gall 2011).

3. The analysis has to describe the data it produces with one or more ontologies. If they are not part of *SEON* yet, they have to be created and linked, if possible, to *SEON*'s core ontologies.
4. Define the web service description using our annotation-enriched WADL, making sure that the service's output and input definitions are, when needed, annotated with links to the ontologies used to represent them.
5. Register the service by submitting the service description created in the previous step to the *SA-B* using its REST API. The *SA-B* will then automatically handle the actual service registration, updating the *Services Catalog* and making the new analysis available to *SOFAS*' users.

6 Applications of software analysis composition

In the following, we present two applications of *SOFAS* analysis composition. The first one is the use of the *SA-B* web UI by human users to define analysis workflows to answer specific software evolution questions. The second one is the use of the *SA-B* RESTful endpoints by a tool called *Software Evolution Perspectives*, to execute a workflow for extracting and visualizing evolutionary data in various perspectives.

6.1 Investigating evolution anomalies with analysis workflows

As a use case for this first type of application of *SOFAS*, we show how we can answer the question "*Which are the hotspots and evolution anomalies for a project?*" by composing a specific workflow using the *SA-B* web UI. Figure 4 shows the UI being used to compose this very workflow. This question and the associated workflow originate from a concrete need we encountered while performing a software quality audit of a commercial software. This software, which we will call Andromeda (due to confidentiality obligations we cannot disclose its real name), is a mission critical system in the domain of facility management for monitoring and maintaining buildings.

We consider a hotspot any source code entity (file, class, method, etc.) that is out of the norm according to different heuristics. Studies (D'Ambros et al. 2009b; Basili et al. 1996) have shown evidence that these anomalies can have a negative effect on software quality, often leading to faults and defects, code brittleness and maintainability issues. Different strategies have been devised to spot them to then support and drive the reengineering process. Some used empirically validated object-oriented source code metrics (Gyimothy et al. 2005; Basili et al. 1996), others a combination of them to find more complex and higher-level problems known as "code disharmonies" (Lanza and Marinescu 2005); others used change couplings extracted from the project revision history (D'Ambros et al. 2009b), etc. All these approaches are valid, however, they do not necessarily find the same hotspots. For example, metrics-based solutions would mainly find code quality related anomalies, while an analysis working on the change couplings between files would find more high level issues such as cross cutting concerns.

With *SOFAS* we can compose workflows that combine some of these different strategies to find a broader spectrum of hotspots and to also find "super hotspots":

entities that present anomalies found with several of the strategies used. The workflow we present here aims to answer the main, generic question by answering and combining four, more specific sub-questions:

1. *Which entities have high or abnormal values of known code quality metrics?*
2. *Which entities present specific code smells?*
3. *Which entities have a high change coupling?*
4. *Which entities have a lot of copied code (code clones)?*

The results are then aggregated to find classes that exhibit all "symptoms," which is the final question: *which entities are super hotspots?* Figure 5 provides an overview of it. As first step, the version history of the project to study is extracted from the associated repository, along with necessary information about the repository. The data produced by this version history service is then fed into a service calculating the project's change couplings (how frequently a class has been changed together with other classes over the evolution of a project).

The *SCoLa* code snippet in Fig. 6 shows how this connection is defined in the language. The two services in the snippet, as the workflow itself, are abstract. This is because the version history service has to be picked at runtime, according to the specific repository used by the project. The change coupling is also left to be instantiated at runtime to further fine tune the analysis with a selection of services implementing different detection strategies. The results of the change couplings service are further refined with a SPARQL query to extract the 10 most significant classes found and answer the analysis question 3. These are the classes with the highest number of coupled classes (NOCC) and sum of coupling (SOC) metrics. The data produced by the version history service is also fed to a SPARQL query to find the most recent release. This is then fed to a service extracting all its code clones and to a service extracting its static source code structure model.

The results of the code clones analysis are fed to a SPARQL query to extract the 10 classes with the highest number of clones and thus answer the analysis question 4. The source code model extracted is then fed to two metrics services. One extracting the most common object oriented metrics and the other one extracting LOC and control flow-related metrics: McCabe's cyclomatic complexity (McCabe 1976), weighted methods per class (WMC) (Chidamber and Kemerer 1994), etc. The results are input to a service that aggregates them and finds code disharmonies, as proposed Lanza and Marinescu (2005). This answers analysis question 2. The results are refined with SPARQL queries to extract the metrics that in studies of Basili et al. (1996) or Gyimothy et al. (2005) have been found to be relevant for defect prediction. So we can answer analysis question 1. Finally, all the results of these four sub-questions are aggregated to find any possible "super hotspot." All the results are also saved, so that the user can eventually fetch and analyze them upon the workflow completion. The *SCoLa* snippet in Fig. 7 shows how this aggregation is defined.

Notice, however, that the services do not share the actual data, which in the case of versioning data could even be in the range of gigabytes. Only the analysis' URL is, and through that, data can be selectively fetched is using *SOFAS*' uniform RESTful interface (as described in Sect. 2). Thus the traffic between the services is kept to a minimum. It is up to the individual services to get the necessary data whenever it is needed.

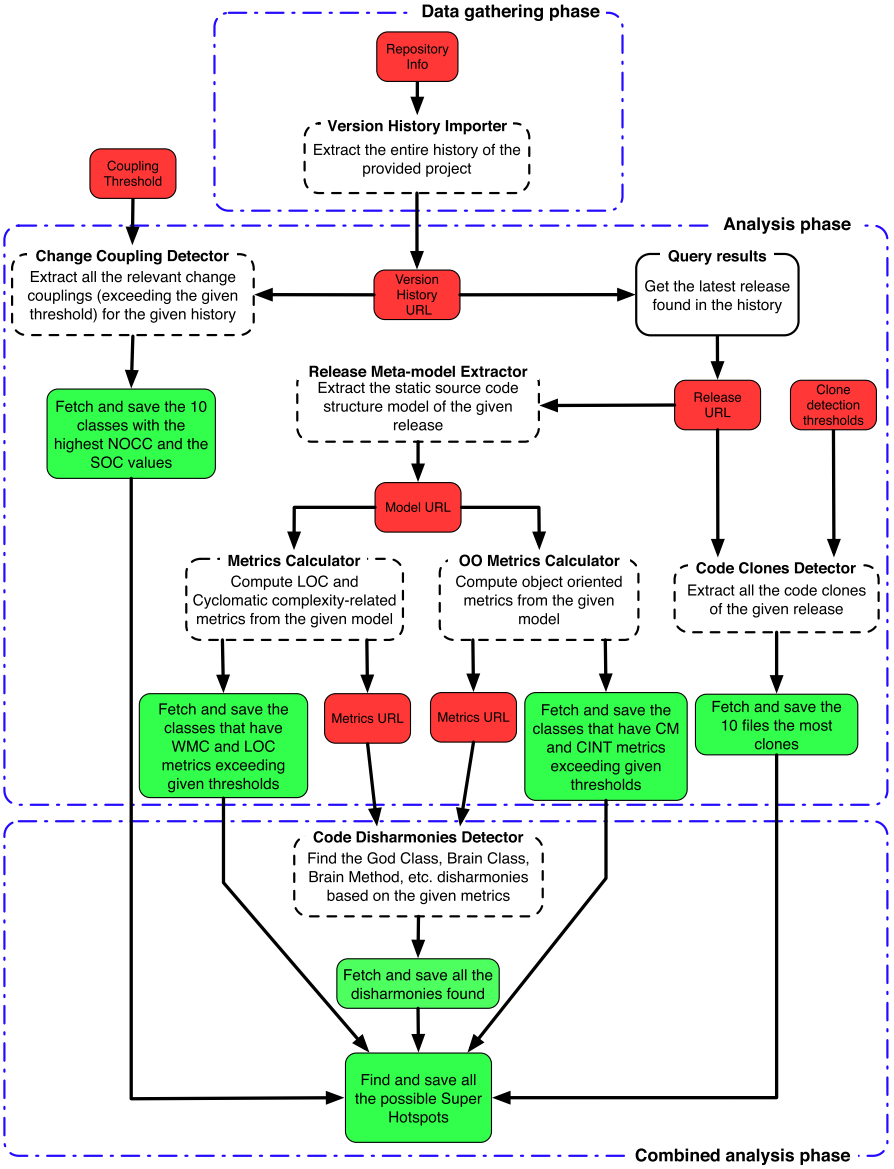

Fig. 5 Overall view of the Hotspots workflow

This workflow has been run for the Andromeda system during a software quality assessment process. In addition to that, it has been ran as a use case validation for some of the most popular Apache Commons[11] projects: commonsValidator,

[11] http://commons.apache.org.

```
<links>
      <link name="versionHistory_to_changeCoupling"/>
      <link name="versionHistory_to_fetchReleaseQuery"/>
</links>
<sequence>
      <invoke analysis="http://habanero.ifi.uzh.ch/sofas/abstractVersionHistory"
            inputVariable="repositoryData"
            outputVariable="historyUrl">
         <sources>
            <source linkName="versionHistory_to_changeCoupling"/>
            <source linkName="versionHistory_to_fetchReleaseQuery"/>
         </sources>
      </invoke>
      <assign>
         <copy>
            <from>$historyUrl</from>
            <to>$changeCouplingInput/customerInfo</to>
         </copy>
         <copy>
            <from>$historyUrl</from>
            <to>$inputUrl</to>
         </copy>
      </assign>
      <flow>
         <invoke analysis="http://habanero.ifi.uzh.ch/sofas/abstractChangeCoupling"
               inputVariable="changeCouplingInput">
               <targets>
                  <target linkName="versionHistory_to_changeCoupling"/>
               </targets>
         </invoke>
         <query sparqlQuery="SELECT ?x WHERE ?x rdf:type v:Release . . . . . "
               inputVariable="inputUrl"
               outputVariable="lastReleaseUrl">
               <targets>
                  <target linkName="versionHistory_to_fetchReleaseQuery"/>
               </targets>
         </query>
      </flow>
</sequence>
```

Fig. 6 A snippet of the actual *SCoLa* definition of the Hotspots workflow as created by the *SA-B* web UI

commonsTransaction, commonsMath, commonsLang, commonsIo, commonsCollections, commonsCodec and commonsCli.

Tables 1 and 2 summarize the results for all these projects, grouped by the analysis question they answer. Please note, that due to space limitations, we are showing only the 5 top most relevant files for each analysis question for the analyzed Apache projects analyzed. In some cases, the relevant entities exceeding the required threshold were even less than that (e.g. for Commons Collections and Cli). On the other hand, we show the entire result set for Andromeda. The classes underlined are the "super hotspots" (if any were found). Andromeda's classes have been renamed due to confidentiality needs.

6.2 Software evolution perspectives with analysis workflows

Analysis workflows such as the one presented in the previous section are extremely valuable in answering specific evolution analysis questions and in singling out, unequivocally, noteworthy entities. However, when used by themselves, they lack the capability to fulfill broader and more open-ended information needs. For example, giving an overall view of the evolution or the current state of a software project or

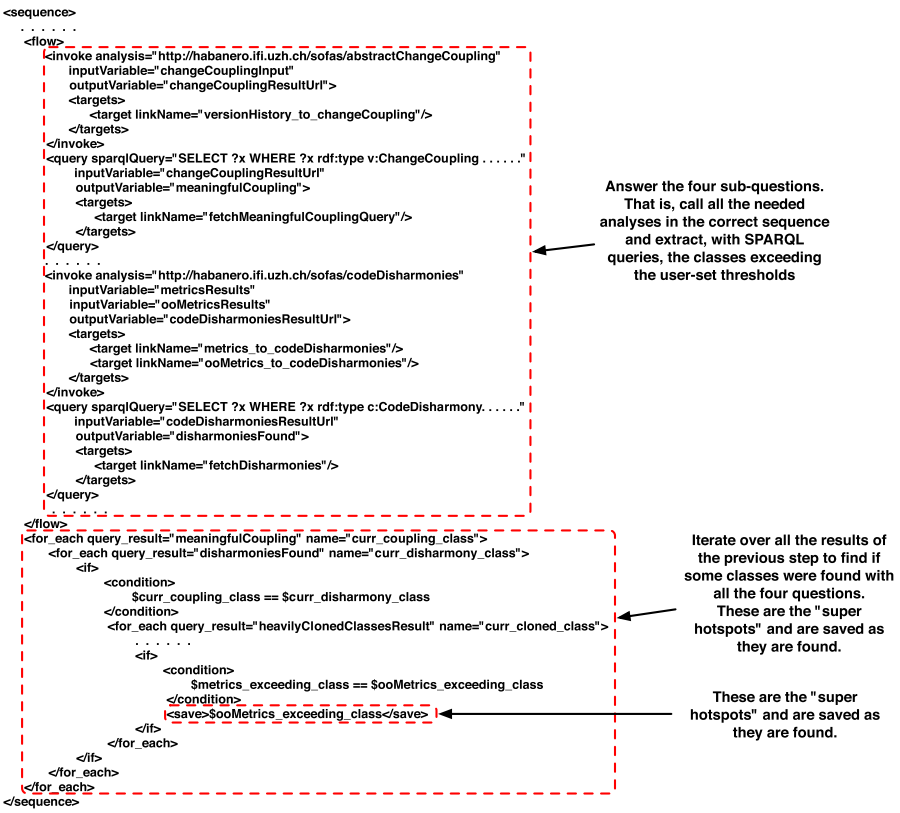

Fig. 7 A snippet of the *SCoLa* definition of the Hotspots workflow showing how results are aggregated and saved

showing trends of specific, critical metrics. They can still provide all the information needed to fulfill those needs but, in this case, human interpretation is heavily needed to put everything into context and draw meaningful conclusions. Our web application *Software Evolution Perspectives* aims exactly at filling this gap.

The main purpose of this tool is to give software evolution and quality analysts a detailed and intuitive overview on the quality of a software project and its history. This is achieved through the use and combination of different "perspectives", focusing on different aspects of the software analyzed. Every perspective offers different interactive visualizations of the aspect addressed, along with automatically generated considerations about it. These considerations are used to better explain the different visualizations and to put them into a software quality context. Moreover, they also explain, for example, why specific results and values found are good/bad for the quality of the software analyzed. The perspectives offered so far are:

Metrics perspective It addresses the visualization and interpretation of the metrics calculated for every release of the analyzed software project—and the code smells detected based on those metrics. These visualizations include:

Table 1 Project Hotspots workflow answers to the first and second question

Project	Which entities have a high change coupling?	Which entities have a lot of duplicate code (code clones)?
Andromeda	*Eve.java*	*Eve.java*
	Fdd.java	*Con.java*
	Con.java	Por.java
	Por.java	Use.java
	Bas.java	
Commons validator	ValidatorResources.java	*Field.java*
	Validator.java	DateValidator.java
	Field.java	EmailValidator.java
	Form.java	UrlValidator.java
	ValidatorAction.java	ValidatorAction.java
Commons Math	EmpiricalDistributionImpl.java	ComposableFunction.java
	GammaDistributionImpl.java	MathUtils.java
	RealMatrixImpl.java	OpenIntToDoubleHashMap.java
	AbstractContinuousDistribution.java	AbstractRealMatrix.java
	ExponentialDistributionImpl.java	RealMatrixImpl.java
Commons IO	AndFileFilter.java	FileUtils.java
	OrFileFilter.java	*NameFileFilter.java*
	PrefixFileFilter.java	IOUtils.java
	NameFileFilter.java	FileWriterWithEncoding.java
	SuffixFileFilter.java	Tailer.java
Commons Codec	Base64Test.java	DoubleMetaphone.java
	Base64.java	RefinedSoundex.java
	RefinedSoundex.java	QuotedPrintableCodec.java
	Metaphone.java	URLCodec.java
	URLCodec.java	Hex.java
Commons Cli	Option.java	OptionBuilder.java
	CommandLine.java	
	Options.java	
	PosixParser.java	
	GnuParser.java	
Commons Collections	CollectionUtils.java	
	MapUtils.java	
	ListUtils.java	
	BufferUtils.java	
	SetUtils.java	
Commons Transaction	GenericLock.java	*FileResourceManager.java*
	GenericLockManager.java	TransactionalMapWrapper.java
	FileResourceManager.java	GenericLockManager.java
	LockManager.java	AbstractXAResource.java
	ResourceManager.java	FileHelper.java

Table 2 Project Hotspots workflow answers to the third and fourth question

Project	Which entities present specific code smells?	Which entities have high or abnormal values of known code quality metrics?
Andromeda	*Eve.java*	Cons.java
	Fdd.java	*Eve.java*
	Com.java	*Con.java*
	Obj.java	Com.java
	Poi.java	Evn.java
	Con.java	Jta.java
	Obi.java	Obi.java
		Rea.java
		Wat.java
		Dow.java
Commons Validator	*Field.java*	*Field.java*
Commons Math	TransformerMap.java	StatUtils.java
		DefaultTransformer.java
Commons IO		*NameFileFilter.java*
		IOUtils.java
		XmlStreamReader.java
Commons Codec		
Commons Cli		
Commons Collections		
Commons Transaction	GenericLock.java	*FileResourceManager.java*

- The metrics pyramid (Lanza and Marinescu 2005) of every release.
- Interactive, navigable kiviat diagrams of all the packages and classes of the system, as proposed by Pinzger et al. (2005).
- The evolution, over time, of the most important project-wide metrics (e.g. LOC, average cyclomatic complexity, etc.).
- A browsable list of all the disharmonies and the code entities (classes and methods) that exhibit them.
- A navigable, interactive treemap of every release of the entire system, with the exceptional entities highlighted so that they can be quickly pinpointed and studied. Exceptional entities are packages or classes that either exhibit values of critical metrics above known thresholds or present specific code disharmonies.

Figures 8 and 9 show a small collection of these visualizations.

Project history perspective It addresses the visualization and interpretation of the history of the project. These visualizations include:

- Interactive, navigable fractals representing the code ownership of every class and package of the project, as proposed by D'Ambros et al. (2005).
- Graphs of some of the most commonly used version control metrics/statistics (i.e. distribution of commits between developers, code churn, etc.).

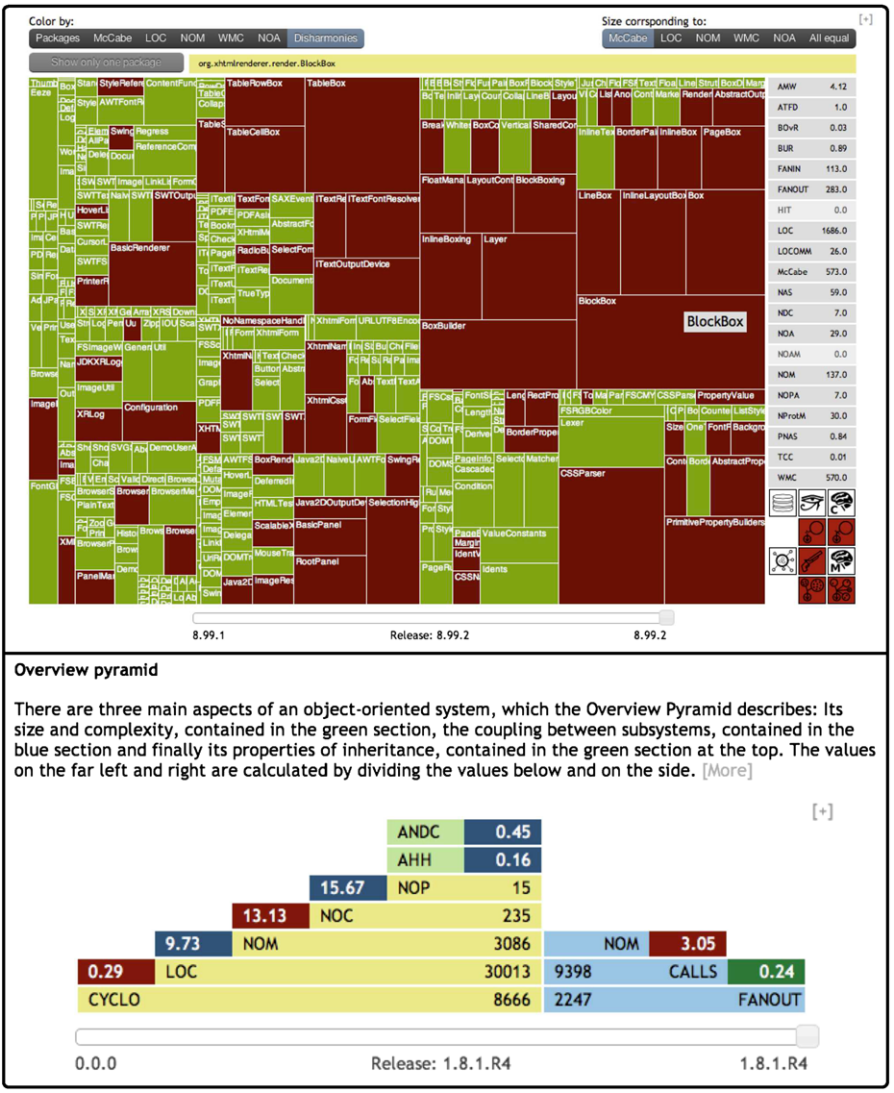

Fig. 8 A screenshot of a metrics pyramid and an interactive kiviat diagram of the Metrics perspective

– Graphs of some of the most commonly used issue tracking metrics/statistics (i.e. bugs open/closed per month, distribution of bugs by different attributes, etc.).

Two of these visualizations are shown in Fig. 10. The first one shows the evolution of the project size during the project's lifetime in terms of total lines of code. The second one shows the monthly distribution of commits between the developers.

Change coupling perspective It addresses the in-depth visualization and interpretation of the logical coupling between entities (files, source code files, modules,

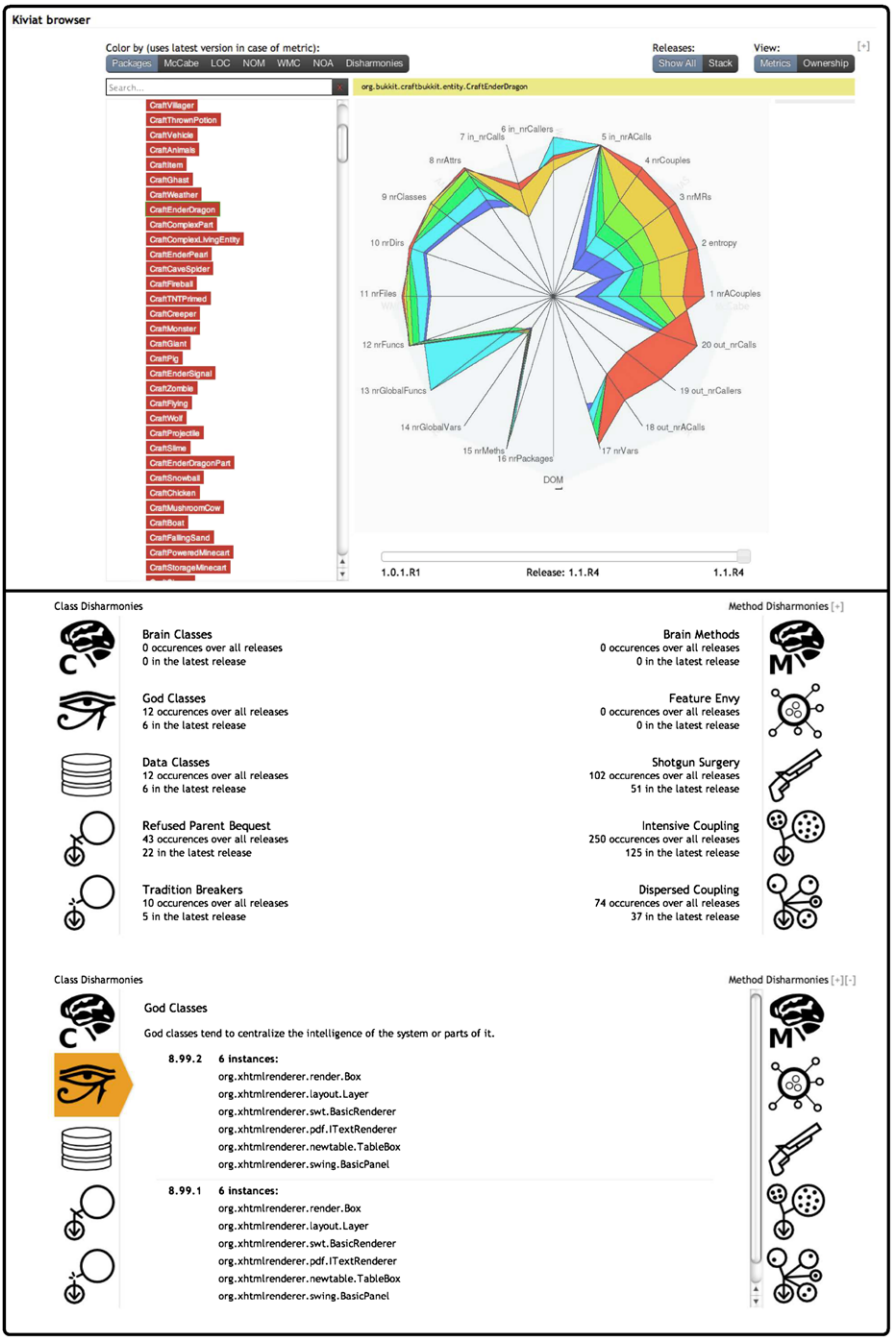

Fig. 9 A screenshot of the navigable, interactive treemap and the disharmonies list of the Metrics perspective

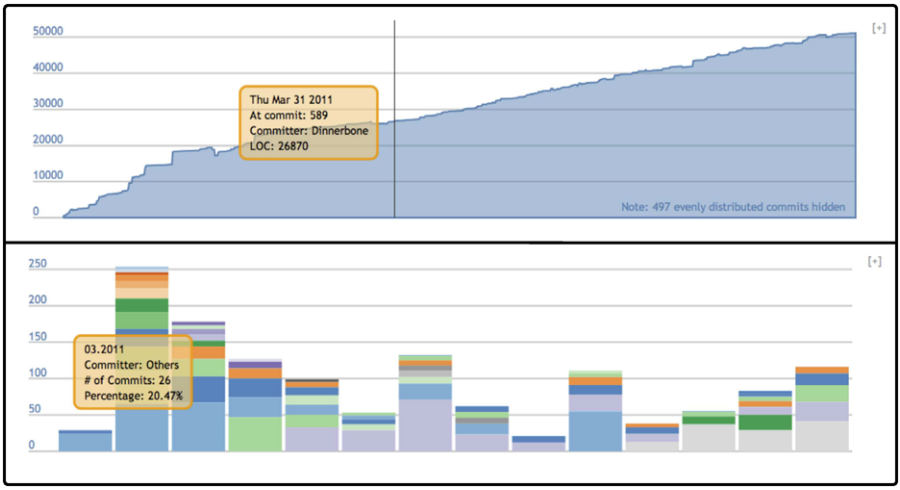

Fig. 10 Two of the visualizations making up the project history perspective

directories, etc.) at different granularity levels, leading to a precise characterization of the system modules in terms of their logical coupling dependencies. It is based on D'Ambros et al.'s (2009a) Evolution Radar.

Fine grained source code changes perspective It gives a detailed view of all the fine grained source code changes that happened throughout the project history. This perspective provides detailed information on the statement and declaration level changes that is missing in the normal change history available in version control system. The visualizations offered include:

- A navigable, interactive change history of every single file, module or the entire system. Showing, for every commit, its overall significance and detailed information on all the associated fine grained changes.
- Pie charts showing the distribution of these changes over the entire history of a single file, module or the entire system.
- A list of the 10 most significant commits in the project history, with details on which files were changed in it and how (with which type of changes).

In addition to these interactive visualizations, *Software Evolution Perspectives* also offers the automatic creation of a software quality and history report based on some of the most relevant visualizations and analyses used in the different perspectives. This report gives a quick overall assessment of the quality of the analyzed project, its hotspots and how these evolved in time. As for the Hotspots workflow we introduced in Sect. 6.1, this was also used in a real software quality audit project with an industrial partner to analyze the Andromeda system.

Software Evolution Perspectives uses a custom, predefined *SCoLa* workflow to combine and execute all the analyses required to get all the data needed by the different perspectives. Figure 11 shows the high-level representation of such workflow. This workflow just combines all the different analyses needed, returning only the URLs to their results. It does not aggregate or work on the results like the one we

presented in Sect. 6.1. This is because, as said, the goal is not to answer a specific question, but to get as much information as possible about the quality and history of the analyzed project. This means that *Software Evolution Perspectives* will then, given those result URLs fetch and aggregate all the data needed directly from the analyses providing them.

Another major difference with the use case presented in the previous section is that, in this case, the actual workflow and *SOFAS* itself are hidden from the user. While to answer specific questions, such as the one in Sect. 6.1, the user uses *SA-B* web UI to compose a suitable workflow, in this case, the tool itself will take care of that. The user will only have to supply the URL of the version control and issue tracking repositories of the project to analyze. The tool then, will automatically detect the systems those URLs refer to, e.g. git, svn, bugzilla, trac, etc., compose a suitable workflow and send it to *SOFAS*' *SA-B* for execution, through its RESTful endpoints. Upon workflow completion, the tool will then fetch all the data needed from the analysis themselves, save it and organize it into the different perspectives.

7 Related work

In this section, we briefly outline some of the major existing works related to our approach. In particular, we discuss the use of ontologies in mining software repositories and software evolution, RESTful web services composition, and tools exploiting and combining historical project data for software evolution analysis.

7.1 Ontologies in software evolution analysis

Several researchers have described software evolution artifacts with OWL ontologies. Their approaches integrated different artifact sources to facilitate analysis activities.

Kiefer et al. proposed EvoOnt, a software repository data exchange format based on OWL (Kiefer et al. 2007). EvoOnt is heavily inspired by Evolizer's (2009) data models and is made up of three sub-ontologies: a software ontology model, a bug ontology model and a version ontology model. The authors used a modified version of SPARQL to detect bad code smells, calculate metrics, and to extract data for visualizing changes in code over time. In an extension to this work, Tappolet et al. (2010) replicated several software evolution and analysis experiments from previous Mining Software Repositories Workshops. As a result, they showed they could replicate 75 % of those analyses with at most two SPARQL queries.

Iqbal et al. (2009) proposed a *Linked Data Driven Software Development* (LD2SD) methodology, which involves transformation of software repository data into RDF format and then indexing with a semantic indexer. The overall goal was to provide a uniform and central RDF-based access to JIRA bug trackers, Subversion, developer blogs, project mailing lists, etc. Integration between the repositories was achieved with *Semantic Pipes*, an RDF-based mashup technology. The results were finally injected into the bug tracker web page, to provide developers with additional, context-related information.

We share with these works the idea of describing varied software repository data with interlinked ontologies. However, none of them organize their ontologies in

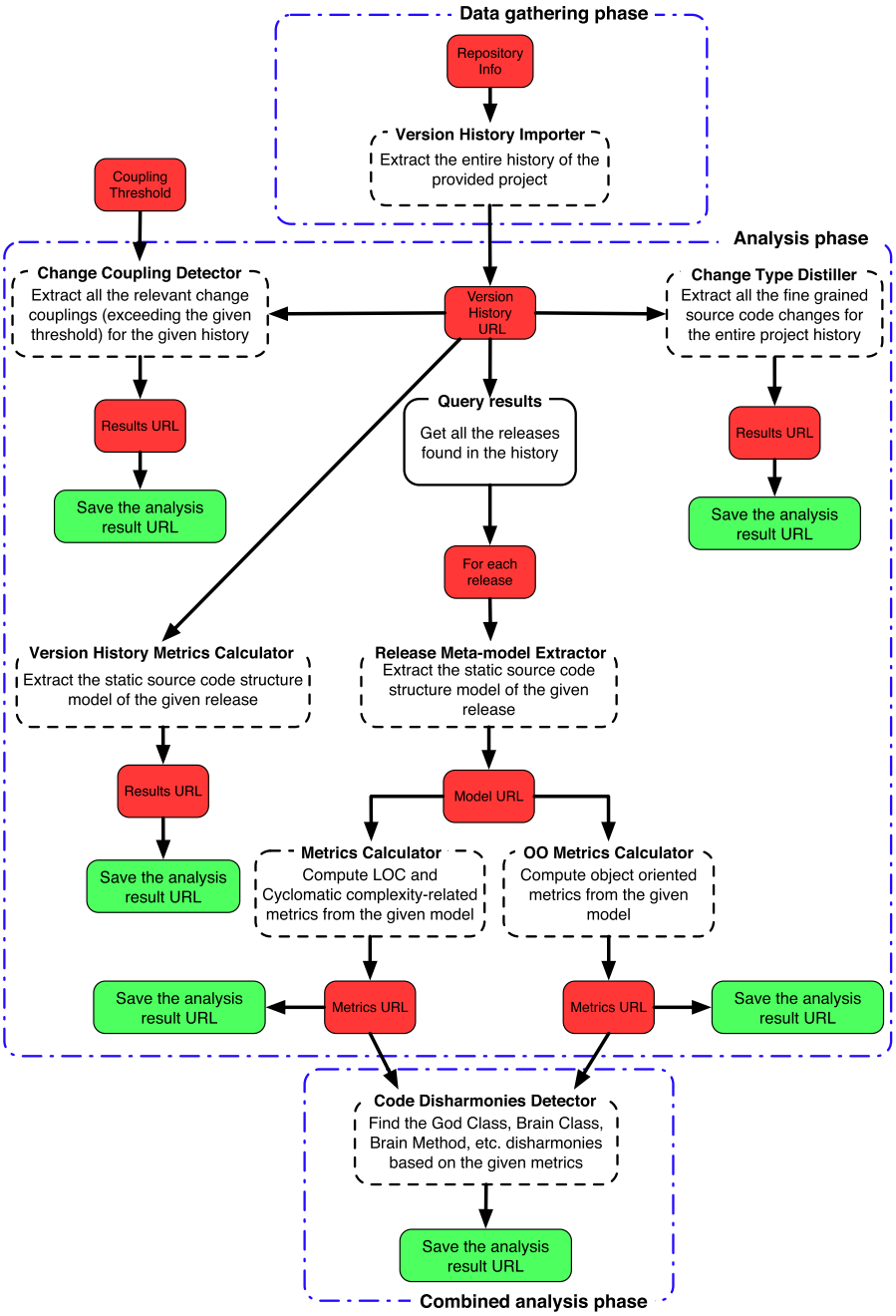

Fig. 11 Overall view of the workflow used by *Software Evolution Perspectives*

consecutive layers of abstractions with clear representational purpose, as we did in *SEON*. Moreover, their main goal is to use these ontologies to make the implicit links between different software artifacts explicit and facilitate the use of such information. While that is also one of our goals, in addition to that, we also use ontologies to promote easier information sharing between analyses and to build more complex and composite analyses on top of this core. At last, these approaches only present a proof of concept of the usefulness of ontologies. A generic framework where data about software artifacts can be automatically collected, analyzed and queried for several purposes is still missing.

7.2 RESTful webservice composition

Web service composition has been throughly addressed–and it still is–in the major software engineering conferences and in the more specific ICSOC and ICWS. A state of the art of is not in the scope of this paper and our work in general. On the other hand, the issue of composing RESTful web services is highly related to our work. Traditionally, these services have been used in a much different context than the traditional SOAP/RPC-based ones. In particular, they have been used as standalone web application or manually, ad-hoc combined with all sort of other services in web 2.0 mashups. The real need for a standard RESTful web service description language and a structured and methodical combination technique has not really come up yet.

Pautasso (2008) and Mandel (2008) both proposed the use of WSDL to describe RESTful web services to then facilitate their composition with other similar services and with classic "big services". Pautasso also introduced, in the same work, an extension to WS-BPEL to natively support REST, without the need of a WSDL bridge.

Other works, such as works of Pautasso (2009), Zhao and Doshi (2009), Mangler et al. (2009), took a more radical approach, proposing the use of new ad-hoc tools (Pautasso 2009) or languages (Zhao and Doshi 2009; Mangler et al. 2009) to describe and compose these services. None of these have really gained much ground or have been used outside theoretical case studies.

Given this situation, it comes as no surprise that more high level concepts as workflow validation or semantic description of web services has not been addressed yet, apart from the recently proposed SA-REST (Lathem et al. 2007). Oddly enough, so far no solution has exploited the already existing WADL, even though it can almost be considered a de-facto standard. On the other hand, we decided to base our solution on WADL as we deemed it mature and expressive enough for our needs. Moreover, it allowed us to use an already existing language without having to define one of our own, which in the end would have had very similar structure and rationale.

7.3 Software evolution analysis composition

There is an abundance of research works and tools exploiting software project data for historical software analysis. The majority of them exploit the source code change history or bug history to study the dynamics underlying software evolution. Only a few address the combination data coming from different analyses and sources.

Systems such as Kenyon (Bevan et al. 2005), Evolizer (Gall et al. 2009) and softChange (German 2004) combine source code change history, bug history and

additional analyses such as, for example, fine grained source code change extraction, source code meta-model reconstruction, etc. All these approaches rely on their own ad-hoc developed tools and techniques and none target the issue of using and composing different, independent analyses. Furthermore, they don't allow the user to combine the analyses into custom combinations. All the supported ones are created beforehand and hardcoded into the tools. At last, all of these approaches are tool-based and none addressed the issue of using web services—or similar technologies—to support and facilitate the analysis usage and composition. OASIS by Jin and Cordy (2005), has been so far the only attempt to find a solution to most of these issues.

We share with OASIS the overall concept, but at the same time, the two approaches have many differences due to their partially distinct goals. Their objective was to allow an analysis available in one tool to use the fact-base of another one in a very simple way. For this reason, they used a domain ontology just to describe the set of representational concepts that the different tools to be integrated require and support. On the other hand, we exploit ontologies on a much broader scale: to catalog and describe the services, to represent and standardize their input and output accordingly to the type of analysis offered, to semantically link different results, to perform (semi)-automatic reasoning on them and, at last, to support the combination of different analyses. In our opinion, this last point is really the most novel and useful feature of our work. At last, their work only sketched the overall rationale of the approach without going into details on how the proposed architecture was actually implemented and which technologies were used. Based on all these considerations, we can claim that the issue of software analysis composition has not yet been systematically tackled.

8 Conclusions

We have investigated the concept of *Software Analysis as a Service*. Such software evolution analyses are offered as services that can be accessed, composed into workflows, and executed over the Internet. This paper described a novel framework for composing such analysis services into workflows, consisting of a custom-made modeling language and a composition infrastructure for the service offerings. The framework exploits the RESTful nature of our analysis service architecture and comes with a service composer to enable semi-automated service compositions by a user. Our workflow language *SCoLa* takes advantage of the RESTful nature of our architecture. It comes with primitive activities such as service invocation and result query, and complex activities such as control flow (sequence, iteration, conditional flow) and parallel execution. Abstract workflows can be defined as templates for repeating service flows, as well as concrete workflows capture specific service compositions.

We validated our framework with an initial set of services which have been developed to fulfill some immediate analysis needs and are mostly based on analyses we previously developed for related projects, such as Evolizer (Gall et al. 2009) and Change Distiller (Fluri et al. 2007). These services helped us populate the framework with enough analyses to provide varied, meaningful and non trivial evolutionary data for a first validation. As proof of concept, we presented two applications of *SCoLa* workflows using these services. Both cases showed the composition of many

different types of analyses into a workflow, but with different purposes. The first application conceptually proves that our framework can be used to address relevant evolution analysis questions, such as finding code locations (i.e. hotspots) that have a high change frequency, intensive change coupling with other entities, and exhibit code clones. The second application shows how tools can harness such workflows to automatically gather a wide range of varied yet interlinked information about a software system and how they can use that for their own specific needs. In our case, we show how this data can be exploited to help stakeholders to gain a better understanding on a software, its history and quality from different perspectives, using intuitive visualizations.

The two applications presented originate from concrete evolution analysis needs we came across in our projects with industrial partners. They demonstrate the usefulness of our approach in answering concrete software evolution and quality questions and information needs. So far these are the only two concrete uses of *SOFAS* combined with *SCoLa*. However, several other tools can be built on top of them and many similar workflows can be defined according to the needs of an analyst. There are many other possible uses of *SOFAS* and *SCoLa* that we intend to explore in the future. For example, we foresee the definition of more ready-to-use workflow blueprints (abstract and concrete) to cover analysis scenarios reported both in the literature and gathered from industrial contexts by means of experiments.

Several *SOFAS* analyses (used without the composition infrastructure presented in this article) have already been proven to be useful to other tools by providing data extracted from a software project. One of them is a Microsoft Surface (now called PixelSense[12]) application that uses the data produced by a single service for purposes of multi-touch enabled code navigation and design recovery (Müller et al. 2012b). Another one, called SMELL TAGGER (Müller et al. 2012a), is a collaborative code review application, that uses the data produced by a combination of *SOFAS* analyses to detect and visualize, on a multitouch screen, the overall code structure, code smells and multiple evolution metrics using different visualization paradigms. Both tools have been used to analyze some of the most popular Java-based open source projects, e.g., ArgoUML, Eclipse, Vuze, jUnit and the entire Apache Software Foundation codebase (more than 300 projects, including Tomcat, Derby, Subversion, Apache HTTPD, etc.). This, in our opinion, is a further testament to *SOFAS* usefulness and flexibility.

Related approaches only allow the combination of analyses into predefined, unmodifiable sequences. Our approach enables users to compose and automatically execute them in a flexible way, based on the particular analysis needs. The composition we devised is only limited by the analyses offered, which is also one of its main weaknesses and threat to validity. In fact, to offer a wide range of potential workflow combinations, a substantial amount of diverse analyses is needed. With the current offering, only workflows working with data extracted from version control systems, issue trackers and plain source code can be created. Moreover, only the analyses offered by the currently registered services (which we introduced in Sect. 2.1) can be ran on that extracted data.

[12] http://www.microsoft.com/en-us/pixelsense.

In the future, we foresee the addition of several other services: from data gatherers to composite analyses targeting diverse evolutionary aspects or offering different algorithms (e.g. other change coupling or code clones detectors). We intend, however, to maintain the focus on software evolution. Thus, we do not plan to add software analyses that are not related to software evolution such as test coverage checks, performance analysis, or control flow analysis. Nevertheless, the analyses currently registered in *SOFAS* are enough to fulfill concrete needs and to showcase the potential of our framework, in particular of its analysis composition features.

Acknowledgements This work was supported by the Swiss National Science Foundation as part of the Systems of Systems Analysis (SoSYA) project, SNF Project No. 132175.

References

Bacchelli, A., Lanza, M., Robbes, R.: Linking e-mails and source code artifacts. In: 32nd International Conference on Software Engineering (ICSE 2010), vol. 1, pp. 375–384 (2010)

Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Softw. **1**(6), 219–232 (2007)

Basili, V., Briand, L., Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)

Bevan, J., Whitehead, E.J., Jr., Kim, S., Godfrey, M.: Facilitating software evolution research with Kenyon. In: ESEC/SIGSOFT FSE, pp. 177–186. ACM, New York (2005)

Business Process Model and Notation (BPMN). OMG standard (2011). http://www.omg.org/spec/BPMN/2.0/

Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)

Čubranić, D., Murphy, G.C.: Hipikat: recommending pertinent software development artifacts. In: Proceedings of the 25th International Conference on Software Engineering, pp. 408–418 (2003)

D'Ambros, M., Lanza, M., Gall, H.: Fractal figures: visualizing development effort for CVS entities. In: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (2005)

D'Ambros, M., Lanza, M., Lungu, M.: Visualizing co-change information with the evolution radar. IEEE Trans. Softw. Eng. **99**, 720–735 (2009a)

D'Ambros, M., Lanza, M., Robbes, R.: On the relationship between change coupling and software defects. In: Working Conference on Reverse Engineering (WCRE '09), pp. 135–144 (2009b)

D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 31–41 (2010)

Farrell, J., Lausen, H.: Semantic annotations for WSDL and XML schema. W3C recommendation, 28 August 2007. http://www.w3.org/TR/sawsdl/

Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)

Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003), pp. 23–32 (2003)

Fluri, B., Würsch, M., Pinzger, M., Gall, H.C.: Change distilling—tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007)

Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based verification of web service compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 152–161 (2003)

Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE 2003), pp. 13–23 (2003)

Gall, H.C., Fluri, B., Pinzger, M.: Change analysis with evolizer and ChangeDistiller. IEEE Softw. **26**(1), 26–33 (2009)

German, D.: Mining CVS repositories, the softChange experience. In: Proceedings of the International Workshop on Mining Software Repositories (MSR), pp. 17–21 (2004)
Ghezzi, G., Gall, H.C.: Towards software analysis as a service. In: Proceedings of the 4th Intl. ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering, L'Aquila, Italy, September 2008. IEEE Comput. Soc., Los Alamitos (2008)
Ghezzi, G., Gall, H.C.: SOFAS: a lightweight architecture for software analysis as a service. In: Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), Boulder, Colorado, USA. IEEE Comput. Soc., Los Alamitos (2011)
Giger, E., Pinzger, M., Gall, H.: Using the Gini coefficient for bug prediction in eclipse. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, pp. 51–55 (2011)
Gini, C.: Variabilità e Mutabilità: Memorie di Metodologica Statistica (1912)
Girba, T., Ducasse, S., Lanza, M.: Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In: Proceedings of the 20th IEEE International Conference on Software Maintenance, pp. 40–49. IEEE Comput. Soc., Los Alamitos (2004)
Gruber, T.R.: A translation approach to portable ontology specifications. Knowl. Acquis. **5**(2), 199–220 (1993)
Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**, 897–910 (2005)
Hughes, G., Bultan, T., Alkhalaf, M.: Client and server verification for web services using interface grammars. In: Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, pp. 40–46. ACM, New York (2008)
Iqbal, A., Ureche, O., Hausenblas, M., Tummarello, G.: LD2SD: linked data driven software development. In: Proc. Int'l Conf. Softw. Eng. and Knowl. Eng (2009)
Jin, D., Cordy, J.R.: Ontology-based software analysis and reengineering tool integration: the OASIS service-sharing methodology. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), pp. 613–616 (2005)
Jordan, D., Evdemon, J.: Web services business process execution language version 2.0. OASIS. Standard, 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
Kiefer, C., Bernstein, A., Tappolet, J.: Mining software repositories with isparql and a software evolution ontology. In: Proceedings of the 4th International Workshop on Mining Software Repositories (MSR 2007) (2007)
Kim, S., Zimmermann, E.W., Jr., Zeller, A.: Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 489–498 (2007)
Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer, New York (2005)
Lathem, J., Gomadam, K., Sheth, A.: SA-REST and (S)mashups: adding semantics to RESTful services. In: International Conference on Semantic Computing (ICSC 2007), pp. 469–476 (2007)
Mandel, L.: Describe REST web services with WSDL 2.0, May 2008. http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/
Mangler, J., Schikuta, E., Witzany, C.: Quo vadis interface definition languages? Towards a interface definition language for restful services. In: International Conference on Service-Oriented Computing and Applications (SOCA 2009), pp. 1–4 (2009)
McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**, 308–320 (1976)
Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: Proceedings of the 8th International Conference on Software Maintenance, pp. 120–130 (2000)
Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering, pp. 181–190 (2008)
Müller, S., Fritz, T., Gall, H.C., Würsch, M.: An approach for collaborative code reviews using multi-touch technology. In: 5th International Workshop on Cooperative and Human Aspects of Software Engineering (2012a)
Müller, S., Würsch, M., Schöni, P., Ghezzi, G., Giger, E., Gall, H.C.: Tangible software modeling with multi-touch technology. In: 5th International Workshop on Cooperative and Human Aspects of Software Engineering (2012b)
Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), pp. 284–292 (2005)

Pautasso, C.: BPEL for REST. In: 7th International Conference on Business Process Management (BPM08) (2008)

Pautasso, C.: Composing RESTful services with JOpera. In: International Conference on Software Composition, Zurich, Switzerland, July 2009, pp. 142–159. Springer, Berlin (2009)

Pinzger, M., Gall, H., Fischer, M., Lanza, M.: Visualizing multiple evolution metrics. In: Proceedings of the ACM 2005 Symposium on Software Visualization (2005)

Robles, G.: Replicating MSR: a study of the potential replicability of papers published in the mining software repositories proceedings. In: 7th IEEE Working Conference on Mining Software Repositories, pp. 171–180 (2010)

Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the International Workshop on Mining Software Repositories (MSR 2005) (2005)

Tappolet, J., Kiefer, C., Bernstein, A.: Semantic web enabled software analysis. J. Web Semant. **8**(2–3), 225–240 (2010)

Tichelaar, S., Ducasse, S., Demeyer, S.: FAMIX and XMI. In: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), Washington, DC, USA, p. 296. IEEE Comput. Soc., Los Alamitos (2000)

Web services choreography description language. 9 November 2005, http://www.w3.org/TR/ws-cdl-10/

Web services business process execution language version 2.0. 8 August 2002, http://www.w3.org/TR/wsci/

Würsch, M., Ghezzi, G., Hert, M., Reif, G., Gall, H.: SEON a pyramid of ontologies for software evolution and its applications. Comput. J., 1–31 (2012)

Xu, C., Qu, W., Wang, H., Wang, Z., Ban, X.: A Petri net-based method for data validation of web services composition. In: IEEE 34th Annual Computer Software and Applications Conference (COMPSAC 2010), pp. 468–476 (2010)

Zhao, H., Doshi, P.: Towards automated RESTful web service composition. In: International Conference on Web Services (ICWS 2009), pp. 189–196 (2009)

Zimmermann, T., Weissgerber, P., Diehl, S., Zeller, A.: Mining version history to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pp. 563–572 (2004)